Experiment 1

AIM:

To install a Unix/Linux operating system on a computer system and verify a successful boot.

Theory:

Linux is a family of free and open-source Unix-like operating systems that follow a modular design: the kernel manages hardware resources, while user-space tools provide shells, package management, and services. Popular distributions (Ubuntu, Fedora, Debian) package the kernel with GNU utilities and a curated software repository, making installation approachable for desktops, servers, and embedded devices.

A typical installation workflow includes preparing bootable media, configuring firmware settings (UEFI/Legacy, Secure Boot), selecting a target disk and partition scheme (GPT with EFI System Partition on UEFI systems), and installing a base system with essential packages. Post-install tasks include setting up users, enabling networking, and applying updates. Verification is done by checking kernel version, boot logs, and package manager health.

A careful installation ensures data safety (backups), correct partition alignment, and a consistent bootloader setup (GRUB/systemd-boot). For dual-boot, one must keep Windows BitLocker/fast-startup considerations in mind and ensure the EFI partition is shared but not overwritten incorrectly.

Program:

```
root@localhost:~$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda      8:0    0 476.9G  0 disk
├─sda1   8:1    0   512M  0 part /boot/efi
├─sda2   8:2    0   100G  0 part /
└─sda3   8:3    0 376.4G  0 part /home

root@localhost:~$ sudo fwupdmgr get-devices
… (UEFI firmware details) …
```

```
root@localhost:~$ uname -r

5.15.0-78-generic



root@localhost:~$ cat /etc/os-release | grep -E 'NAME=|VERSION='

NAME="Ubuntu"

VERSION="22.04.4 LTS (Jammy Jellyfish)"



root@localhost:~$ sudo apt update && sudo apt -y upgrade

… package index updates and upgrades …
```

Conclusion:

A Linux distribution was installed successfully with a proper partition layout and bootloader configuration. System integrity was verified by checking kernel version, OS release, and performing updates.

Experiment 2

AIM:

To study user login/logout information and active session details in Unix/Linux.

Theory:

Unix/Linux tracks user sessions via structured log files and in-memory records. The utmp database (often /var/run/utmp) stores information about current logins; wtmp (/var/log/wtmp) maintains historical records of logins, logouts, and system boots; btmp (/var/log/btmp) captures failed login attempts. Tools such as who, w, last, and lastb present these records for monitoring and auditing purposes.

Understanding these sources is critical for security and operations: administrators can identify suspicious access, trace user activity windows, and correlate load averages with user-initiated processes. Finger can show additional account information if configured. Proper log rotation and permissions ensure privacy and performance.

In systemd-based distributions, journalctl further complements classic files by aggregating boot and service logs. Combining session data with process listings (ps, top) provides a holistic view of who is logged in and what they are running.

Program:

```
root@localhost:~$ who

aryan     tty7         2025-09-23 07:15 (:0)

root      pts/0        2025-09-23 06:58 (192.168.1.5)


root@localhost:~$ w

 07:20:35 up  1:23,  2 users,  load average: 0.15, 0.07, 0.05

USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT

akhil    tty7     :0               07:15    1:23   0.30s  0.05s
/usr/bin/bash

root     pts/0    192.168.1.5      06:58    0.00s  0.02s  0.01s who


root@localhost:~$ last | head -n 5
```

```
akhil    tty7         :0               Tue Sep 23 07:15   still logged in
root     pts/0        192.168.1.5      Tue Sep 23 06:58 - 07:19  (00:21)
reboot   system boot  5.15.0-78-generic Tue Sep 23 06:55   still running


root@localhost:~$ sudo lastb | head -n 3
btmp begins Tue Sep 23 06:00:00 2025


root@localhost:~$ finger aryan
Login: akhil                            Name: Akhil
Directory: /home/akhil                  Shell: /bin/bash
On since Tue Sep 23 07:15 (IST) on tty7 from :0
```

Conclusion:

We examined active and historical session data using who, w, last, and finger. These utilities help correlate user presence with activity for auditing and support.

Experiment 3

AIM:

To explore general purpose Unix/Linux utilities for files, processes, and system information.

Theory:

Core utilities in Linux follow the philosophy of small, composable tools. File and directory operations are handled by ls, cp, mv, rm, mkdir, rmdir, and pwd; inspection and viewing by cat, more/less, and echo; metadata and calendar/time by date and cal; and manual pages by man support self-service learning.

Process introspection uses ps to list processes and kill to send signals. Permissions and ownership are managed via chmod and chown. who and history provide user/session context and command history respectively. shutdown enables controlled system power-off or reboot.

These utilities can be combined with pipes and redirection to accomplish complex tasks succinctly, enabling quick diagnostics and automation in shell scripts.

Program:

```
root@localhost:~$ man ls | head -n 3
LS(1)                          User Commands                          LS(1)
NAME
       ls - list directory contents


root@localhost:~$ ls -l
total 8
-rw-r--r-- 1 akhil akhil  4096 Sep 23 07:10 notes.txt
drwxr-xr-x 2 akhil akhil  4096 Sep 23 07:12 projects


root@localhost:~$ cp notes.txt projects/notes_copy.txt
root@localhost:~$ mv projects/notes_copy.txt projects/notes_backup.txt
root@localhost:~$ chmod 644 projects/notes_backup.txt
```

```
root@localhost:~$ chown akhil:akhil projects/notes_backup.txt



root@localhost:~$ ps aux | head -n 5

root          1  0.0  0.1 167536  9800 ?        Ss   06:55   0:01 /sbin/init

root        532  0.0  0.2 215000 16300 ?        Ss   06:55   0:00
/lib/systemd/systemd-journald

aryan      4321  0.1  0.3  96500 24000 pts/0   Sl+  07:18   0:02
/usr/bin/bash



root@localhost:~$ date

Tue Sep 23 07:21:30 IST 2025



root@localhost:~$ cal Sep 2025

   September 2025

Su Mo Tu We Th Fr Sa

    1  2  3  4  5  6

 7  8  9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30



root@localhost:~$ shutdown -r +5 "Rebooting for maintenance"

Shutdown scheduled for Tue 2025-09-23 07:26:30 IST, use 'shutdown -c' to
cancel.
```

Conclusion:

Common utilities for file, process, and system management were demonstrated. Mastery of these tools is foundational for daily Linux administration and scripting.

Experiment 4

AIM:

To study the vi editor, its modes, navigation, and essential commands.

Theory:

vi (and its improved variant vim) is a modal text editor with two primary modes: Normal (command/navigation) and Insert (text entry). Normal mode supports efficient movement, selection, search, and editing, while Insert mode allows free typing. Ex commands (accessed with :) handle saving, quitting, search/replace, and configuration.

Core motions (h/j/k/l, w/W, b/B, gg/G) and operators (d, c, y, p) compose powerful edits. Visual mode allows block selections; search uses /pattern and n/N to navigate matches. The editor is ubiquitous across Unix-like systems, making it invaluable for remote administration and quick edits in terminals.

Customization via ~/.vimrc (or minimal vi settings) can improve usability—for instance, enabling line numbers, syntax highlighting, and indentation rules for programming.

Program:

```
root@localhost:~$ vi sample.txt

-- INSERT --

(type text here)

:wq


root@localhost:~$ cat sample.txt

Hello from vi editor!


root@localhost:~$ vi sample.txt

:%s/Hello/Hi/g

:set number

:wq
```

Conclusion:

We learned vi basics—modes, navigation, and save/quit commands—and performed simple edits and substitutions on a sample file.

Experiment 5

AIM:

To implement Docker on Linux by installing the engine and running a test container.

Theory:

Docker provides OS-level virtualization, packaging applications and dependencies into portable containers. Containers share the host kernel yet maintain isolation through namespaces and cgroups, enabling efficient development, CI/CD, and microservices deployment.

A standard setup includes installing the Docker engine, enabling the service, and managing images/containers with the docker CLI. Pulling images from registries (Docker Hub) and running containers (docker run) demonstrates the workflow. Non-root access can be configured by adding users to the docker group.

Key concepts include images (immutable layers), containers (runtime instances), volumes (persistent data), and networks (connectivity between containers and services).

Program:

```
root@localhost:~$ sudo apt update && sudo apt install -y docker.io
… installing docker.io and dependencies …

root@localhost:~$ sudo systemctl enable --now docker
root@localhost:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Hello from Docker!

root@localhost:~$ sudo usermod -aG docker aryan
root@localhost:~$ newgrp docker
root@localhost:~$ docker ps
CONTAINER ID   IMAGE         COMMAND   CREATED   STATUS    PORTS   NAMES
```

Conclusion:

Docker was installed and validated by running the hello-world container. We reviewed images, containers, and basic lifecycle commands.

Experiment 6

AIM:

To study Bash shell, Bourne shell (sh), and C shell (csh) in Unix/Linux.

Theory:

Unix shells interpret user commands and provide scripting capabilities. The original Bourne shell (sh) established foundational syntax; Bash (Bourne Again SHell) extends it with command history, better arithmetic, arrays, and improved scripting features. C shell (csh) and tcsh offered a C-like syntax and interactive enhancements historically favored by some users.

Modern systems default to bash or zsh for interactive use, while /bin/sh remains for POSIX-sh compatible scripts. Understanding differences (conditionals, loops, arrays, redirection) helps port scripts and choose the right interpreter for reliability and features.

Program:

```
root@localhost:~$ echo $SHELL
/bin/bash

root@localhost:~$ bash --version | head -n 1
GNU bash, version 5.1.16(1)-release

root@localhost:~$ sh -c 'echo POSIX sh here'
POSIX sh here

root@localhost:~$ csh -c 'echo C shell here'
C shell here
```

Conclusion:

We compared shells and verified available interpreters. Bash is typically default; understanding shell syntax differences is vital for portable scripting.

Experiment 7

AIM:

To study the Unix/Linux filesystem tree structure and standard directories.

Theory:

Linux organizes files in a single rooted hierarchy beginning at /. Core directories include /bin and /sbin for essential binaries, /etc for configuration, /var for variable data (logs, spool), /usr for read-mostly userland, /home for user directories, and /tmp for temporary files. On UEFI systems, /boot/efi hosts the EFI System Partition mount.

The Filesystem Hierarchy Standard (FHS) guides directory purposes and layout. Understanding this structure helps navigate, troubleshoot, and manage storage, backups, and deployments effectively.

Program:

```
root@localhost:~$ ls -1 /
bin
boot
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
```

```
srv

sys

tmp

usr

var


root@localhost:~$ tree -L 1 /

/

├── bin

├── boot

├── dev

├── etc

├── home

├── lib

├── lib64

├── media

├── mnt

├── opt

├── proc

├── root

├── run

├── sbin

├── srv

├── sys

├── tmp

├── usr

└── var
```

Conclusion:

The hierarchical tree rooted at / was reviewed along with primary directories and their purposes per FHS.

Experiment 8

AIM:

To study user and system shell initialization files and environment variables.

Theory:

~/.bashrc is executed for interactive non-login bash shells and is ideal for aliases, functions, and prompt customization. System-wide defaults can be placed in /etc/bash.bashrc or /etc/bashrc depending on the distribution. Login shells source profile files like /etc/profile and ~/.profile.

Environment variables (PATH, HOME, USER, SHELL, LANG) shape program behavior and search paths. Exported variables propagate to child processes, enabling configuration of tools and toolchains consistently across sessions.

Program:

```
root@localhost:~$ head -n 10 ~/.bashrc
alias ll='ls -alF'
PS1='\u@\h:\w\$ '
export EDITOR=vim

root@localhost:~$ grep -E 'PATH|HOME|USER|SHELL' <<< "$(env)" | sort
HOME=/home/akhil
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SHELL=/bin/bash
USER=akhil
```

Conclusion:

We examined startup files and manipulated environment variables, reinforcing where to configure interactive behavior vs. system-wide defaults.

Experiment 9

AIM:

To write a shell script that displays the list of currently logged-in users.

Theory:

The who command lists active sessions from the utmp database. A wrapper script can standardize output, add timestamps, or be scheduled via cron for periodic logging.

Scripting promotes reusability and allows post-processing (filters, notifications) depending on administrative needs.

Program:

```
root@localhost:~$ cat show_logged_in.sh

#!/bin/bash

echo "$(date): Active sessions"

who



root@localhost:~$ chmod +x show_logged_in.sh

root@localhost:~$ ./show_logged_in.sh

Tue Sep 23 07:30:00 IST 2025: Active sessions

akhil     tty7         2025-09-23 07:15 (:0)

root      pts/0        2025-09-23 06:58 (192.168.1.5)
```

Conclusion:

A simple script was created to display current logins, showing how command output can be wrapped and timestamped.

Experiment 10

AIM:

To write a shell script that prints HELLO WORLD to the terminal.

Theory:

Shell scripts begin with a shebang (#!) indicating the interpreter. Simple echo statements demonstrate output and serve as the first step towards more complex automation.

Executable permissions and correct line endings (LF on Unix) are essential for successful execution on Linux.

Program:

```
root@localhost:~$ cat hello.sh

#!/bin/bash

echo "HELLO WORLD"


root@localhost:~$ chmod +x hello.sh

root@localhost:~$ ./hello.sh

HELLO WORLD
```

Conclusion:

The classic HELLO WORLD demonstrated script creation, permissions, and execution via bash.

Experiment 11

AIM:

To develop a shell-based scientific calculator using bc for precision arithmetic.

Theory:

The bc utility provides an arbitrary precision calculator language with functions like s(x), c(x), l(x), e(x), and scaling. Shell scripts can pipe expressions to bc to support floating-point operations beyond the shell's integer arithmetic.

By reading user input and passing it to bc with a defined scale, we can implement a compact yet powerful calculator supporting parentheses and functions.

Program:

```
root@localhost:~$ cat calc.sh

#!/bin/bash

echo "Enter expression (e.g., s(1)+3/2 or (2.5^2)+l(10)) :"

read exp

echo "scale=5; $exp" | bc -l


root@localhost:~$ chmod +x calc.sh

root@localhost:~$ ./calc.sh

Enter expression (e.g., s(1)+3/2 or (2.5^2)+l(10)) :

s(1)+3/2

2.34147
```

Conclusion:

Using bc -l enabled scientific functions and floating-point math. The script reads arbitrary expressions and evaluates them precisely.

Experiment 12

AIM:

To write a shell script that checks whether a given number is even or odd.

Theory:

Even numbers are divisible by 2; modulo operation (%) can be used to test parity. Input validation (ensuring numeric data) improves robustness.

Conditional constructs (if/else) and arithmetic expansion $(( )) are standard in bash for simple numeric logic.

Program:

```
root@localhost:~$ cat even_odd.sh
#!/bin/bash
read -p "Enter a number: " n
if ! [[ $n =~ ^-?[0-9]+$ ]]; then
  echo "Not a valid integer"; exit 1
fi
if [ $((n % 2)) -eq 0 ]; then
  echo "Even"
else
  echo "Odd"
fi


root@localhost:~$ chmod +x even_odd.sh && ./even_odd.sh
Enter a number: 7
Odd
```

Conclusion:

The script correctly determines parity using modulo arithmetic and basic input validation

Experiment 13

AIM:

To write a shell script that searches whether an element is present in a list.

Theory:

Bash supports arrays (read -a) and iteration with for loops. String comparison within a loop enables a linear search. For large data, tools like grep or associative arrays might be preferred.

Proper quoting ("${arr[@]}") preserves elements containing spaces.

Program:

```
root@localhost:~$ cat search_list.sh
#!/bin/bash
echo "Enter space-separated elements:"
read -a arr
read -p "Enter value to search: " val
found=0
for x in "${arr[@]}"; do
  if [ "$x" = "$val" ]; then found=1; break; fi
done
[ $found -eq 1 ] && echo "Found" || echo "Not Found"


root@localhost:~$ chmod +x search_list.sh && ./search_list.sh
Enter space-separated elements:
apple banana cherry
Enter value to search: banana
Found
```

Conclusion:

A linear search over a bash array was implemented, demonstrating user input handling and element comparison.

Experiment 14

AIM:

To write a shell script that checks whether the given path is a directory or not.

Theory:

File tests like -d (directory), -f (regular file), and -e (exists) are built into bash. These guard scripts against invalid paths and enable conditional flows.

Error handling improves UX: prompting the user again or exiting with an informative message.

Program:

```
root@localhost:~$ cat is_directory.sh
#!/bin/bash
read -p "Enter a path: " p
if [ -d "$p" ]; then
  echo "Directory"
else
  echo "Not a directory"
fi


root@localhost:~$ chmod +x is_directory.sh && ./is_directory.sh
Enter a path: /etc
Directory
```

Conclusion:

Using the -d test operator, the script distinguishes directories from other file types or invalid paths.

Experiment 15

AIM:

To write a shell script that counts the number of entries in a directory.

Theory:

Counting directory entries can be achieved by piping ls -1 to wc -l, or more robustly by using find with -maxdepth 1 to avoid locale/format issues. Filters can restrict to files, directories, or patterns.

Quoting the path prevents word splitting on spaces.

Program:

```
root@localhost:~$ cat count_files.sh

#!/bin/bash

read -p "Enter directory: " d

if [ ! -d "$d" ]; then echo "Invalid directory"; exit 1; fi

count=$(find "$d" -maxdepth 1 -type f | wc -l)

echo "Files: $count"


root@localhost:~$ chmod +x count_files.sh && ./count_files.sh

Enter directory: /var/log

Files: 37
```

Conclusion:

We counted regular files using find for better robustness, illustrating pipelines and command substitution.

Experiment 16

AIM:

To write a shell script that copies the contents of one file to another.

Theory:

The cp utility copies files and directories while preserving permissions with -p if needed. Alternatively, redirection (cat source > target) demonstrates shell I/O semantics.

Validation should check that the source exists and target path is writeable to prevent data loss.

Program:

```
root@localhost:~$ cat copy_file.sh

#!/bin/bash

read -p "Source file: " s

read -p "Target file: " t

[ -f "$s" ] || { echo "Source missing"; exit 1; }

cp "$s" "$t"

echo "Copied $s -> $t"


root@localhost:~$ echo "sample text" > a.txt

root@localhost:~$ chmod +x copy_file.sh && ./copy_file.sh

Source file: a.txt

Target file: b.txt

Copied a.txt -> b.txt

root@localhost:~$ cat b.txt

sample text
```

Conclusion:

File copying was automated with basic checks, reinforcing the use of cp and redirection concepts.

To create a directory, write contents into it, and copy it to a location in the home directory.

Theory:

Directory manipulation with mkdir and recursive copying with cp -r supports simple deployment/readiness tasks. Echo and redirection are quick ways to populate test content.

Using ~ for home ensures portability across accounts; quoting paths avoids issues with spaces.

Program:

```
root@localhost:~$ mkdir project_lab

root@localhost:~$ echo "README for lab" > project_lab/README.txt

root@localhost:~$ cp -r project_lab ~/project_lab_copy

root@localhost:~$ ls -l ~/project_lab_copy

total 4

-rw-r--r-- 1 aryan aryan 17 Sep 23 07:40 README.txt
```

Conclusion:

We created and populated a directory then copied it into the home folder, confirming the expected files were present.

To use a pipeline and command substitution to set the length of a line in a file to a variable.

Theory:

Pipelines connect stdout of one command to stdin of another, enabling streaming transformations. Command substitution $(...) captures command output into variables for reuse in scripts.

wc -c returns byte counts; for line lengths, head -n 1 or sed -n '1p' can isolate a line before counting. Locale and encoding considerations may affect the notion of character vs. byte.

Program:

```
root@localhost:~$ echo "Hello Linux" > line.txt

root@localhost:~$ len=$(head -n 1 line.txt | wc -c)

root@localhost:~$ echo $len

12
```

Conclusion:

We demonstrated $(...) and pipes by capturing the first line's byte length into a shell variable for later logic.

To write a sed command that prints duplicated lines of input.

Theory:

sed streams text through editing rules expressed as commands and regex patterns. Detecting duplicates can be done with hold space, line numbering, or sorting followed by uniq -d. A pure-sed approach typically compares adjacent lines or leverages pattern/hold spaces.

For simplicity, pre-sorting input ensures duplicates become adjacent; then sed or uniq can reveal duplicated entries. Here, we demonstrate sed printing only repeated lines from a sorted stream.

Program:

```
root@localhost:~$ cat data.txt
alpha
beta
alpha
gamma
beta

root@localhost:~$ sort data.txt | sed 'N; s/^\(.*\)\n\1$/\1/; t dup; D;
:dup; p; D'
alpha
beta
```

Conclusion:

Using sed on a sorted stream, we emitted only the values that appeared consecutively (duplicates), demonstrating regex back-references and multi-line patterns.